LED CUBE

ECE 405 – Senior Design III
Fall 2012 Semester

Document prepared by
Aaron Aaberg
Jared Gratzek
Dane Swartz

Group Advisor: Professor Mark Schroeder, PhD

Table of Contents

LED CUBE

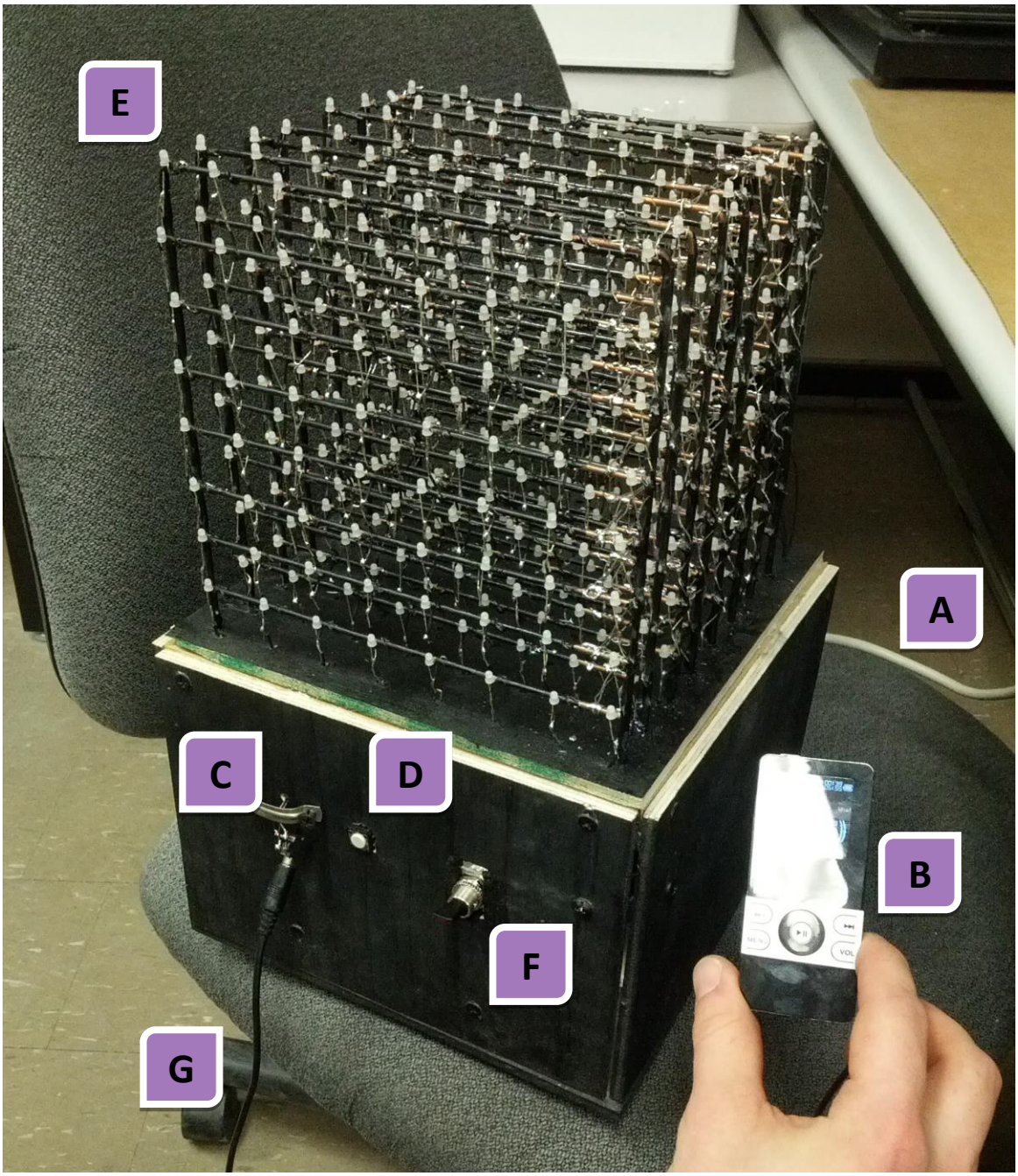
Aaron Aaberg - Jared Gratzek - Dane Swartz

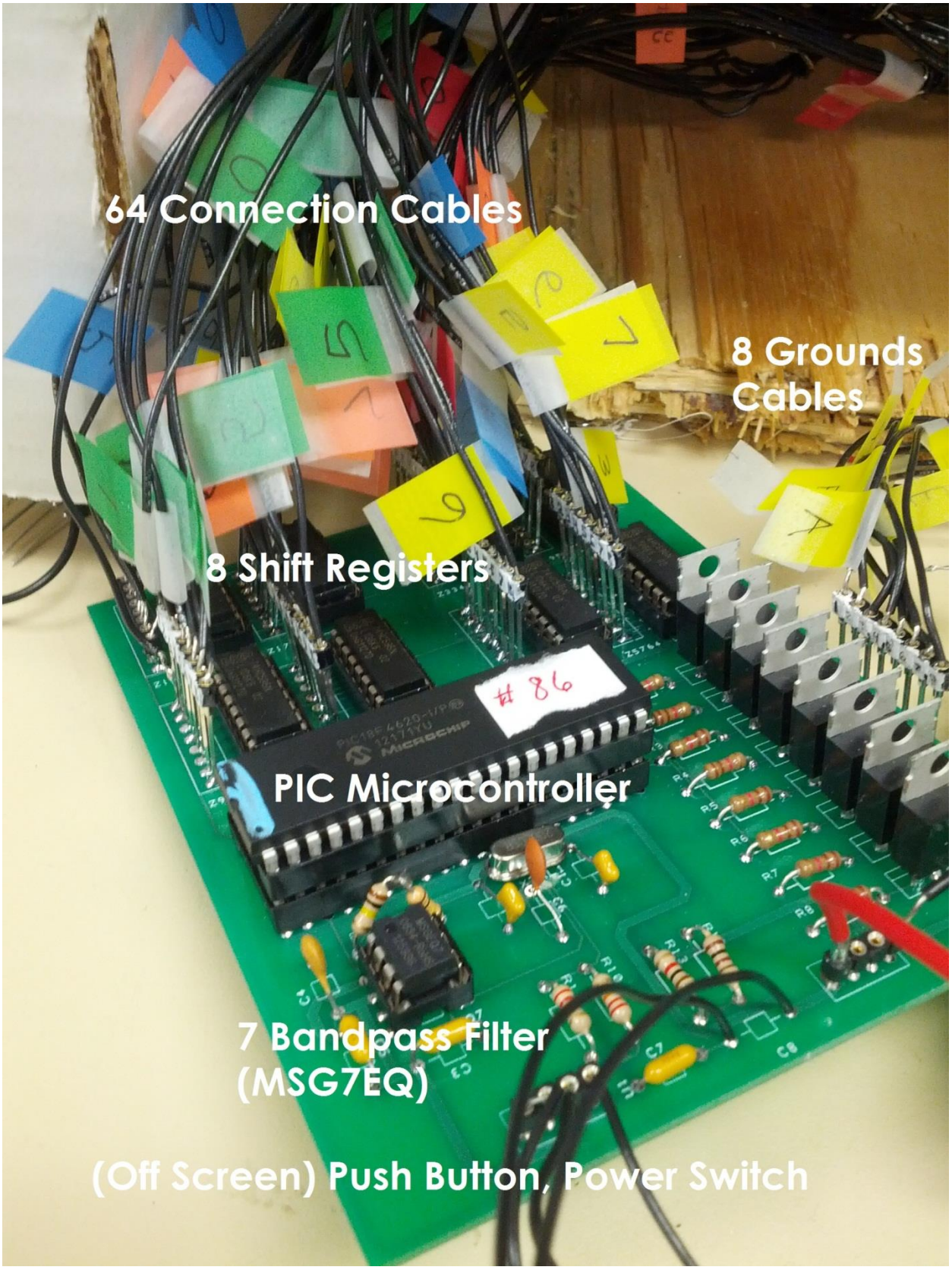
Description:

Back in the 70's there was a novel possession found in almost every best room, this glorious device was the Lava Lamp. This object has had a great impact on pop culture ever since. Fast forward 40 years and Lava Lamps have essentially gone extinct. Now is the time to revive the idea of this eye pleasing lamp and update it with technology from the 21st century. We proudly present the LED Cube.

We've modernize the lava lamp. This device consists of 512 LEDs formed in an 8x8x8 cube. A music source and be sent into the device and transformed into an amazing lightshow for all ages.

(A) **AC Power Cable**
(B) **Music Source (**not included**)**
(C) **3.5mm Auxiliary Cable Port**
(D) **Mode Change Button**
(E) **512 LED, 8x8x8 Cube**
(F) **Plexiglas protective cover**
(G) **3.5mm Auxiliary Cable**




64 Connection Cables
8 Grounds Cables
8 Shift Registers
PIC Microcontroller
7 Bandpass Filter (MSG7EQ)
(Off Screen) Push Button, Power Switch

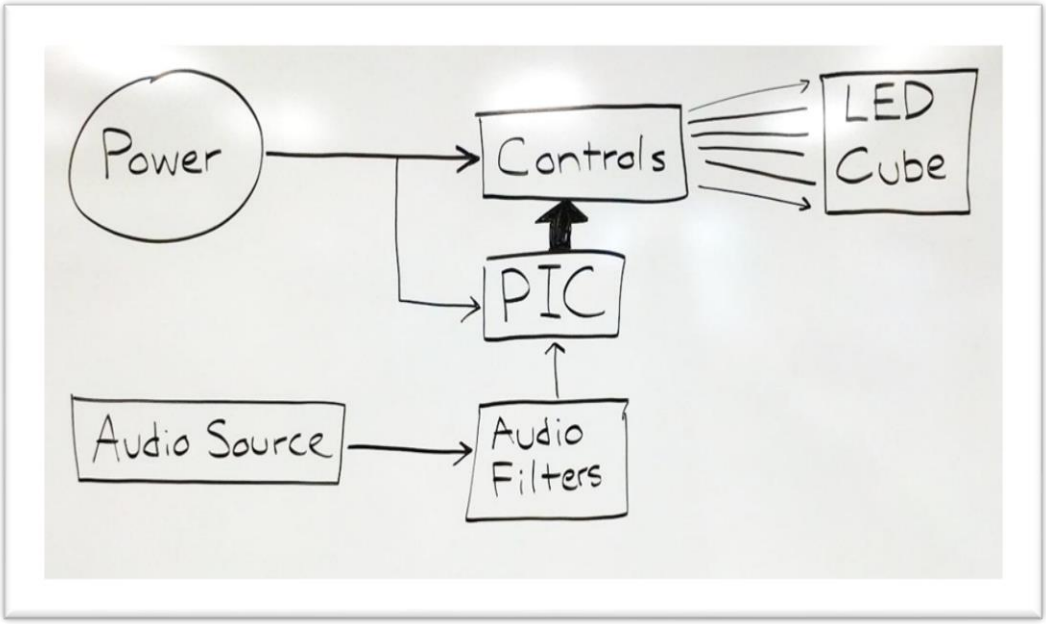


Diagram:

The above diagram shows the audio source's signal (MP3 player, CD player or dubstep synthesizer) will be fed into the device's audio filters. Here the audio will be managed and passed to the PIC for processing. Depending on the data, the PIC will control the LED cube through our control circuitry.

	SUMMER	6-Sep	13-Sep	20-Sep	27-Sep	4-Oct	11-Oct	18-Oct	25-Oct	1-Nov	8-Nov
PHASE NAME	WEEK 0	WEEK 1	WEEK 2	WEEK 3	WEEK 4	WEEK 5	WEEK 6	WEEK 7	WEEK 8	WEEK 9	WEEK 10
CUBE ASSEMBLY											
CREATE CASE											
PIC DESIGN/BUILD											
CONTROL DESIGN/BUILD											
AUDIO FILTERING											
PROGRAMMING PHASE											
FINAL ASSEMBLY											
PRESENTATION PREP.											

Requirements:

- ✓ This device will consist of 512 LEDs formed in an 8x8x8 cube.
- ✓ The LED cube will be assembled using a copper frame. The LEDs will be directly soldered to copper bars which will be fastened together to forming eight layers. These eight layers will be uniformly painted black and assembled identically. Arranging the separate layers together will lead to the 8x8x8 cube.
- ✓ The Cathode post – short leg – will be soldered to the copper bar which will be grounded. The Anode post – long leg – will be power. This scheme will be as followed: columns=power, rows=grounded.
- ✓ The housing for the LED cube will be contained inside of a clear plexiglas case.
- ✓ All resulting circuitry will be out of sight, contained inside of a housing.
- ✓ LEDs can be turned on separately and controlled individually.
- ✓ The cube will be able to response to audio.
- ✓ The LED cube will receive a signal through a 3.5mm stereo audio jack, allowing for multiple source devices (MP3 player, dubstep machine, CD player, etc.).
- ✓ If the device is turned on and no audio signal is present, the LED cube will go into a preprogrammed effect mode.

LED Driver Connection Scheme:

Objective: The objective of the electronic hardware will be to have individual control of the 512 LEDs. In order to minimize the number of traces from the LEDs, a unique wiring scheme was implemented instead of each LED having its own power and ground.

The wiring scheme that will be implemented is in the following figure. All of the anodes within each of the 64 columns (8 LEDs per column) are connected together and share power. All of the cathodes within each of the 8 rows (64 LEDs per row) will be connected together and share grounds.

Total there will be 64 'power' columns and 8 'ground' rows. This scheme allows for individual control of each LED and reduces the amount of connections from 1024 (512 power, 512 grounds) to 72 connections (64 power, 8 grounds).

Control Examples:

To turn on just the top left LED:

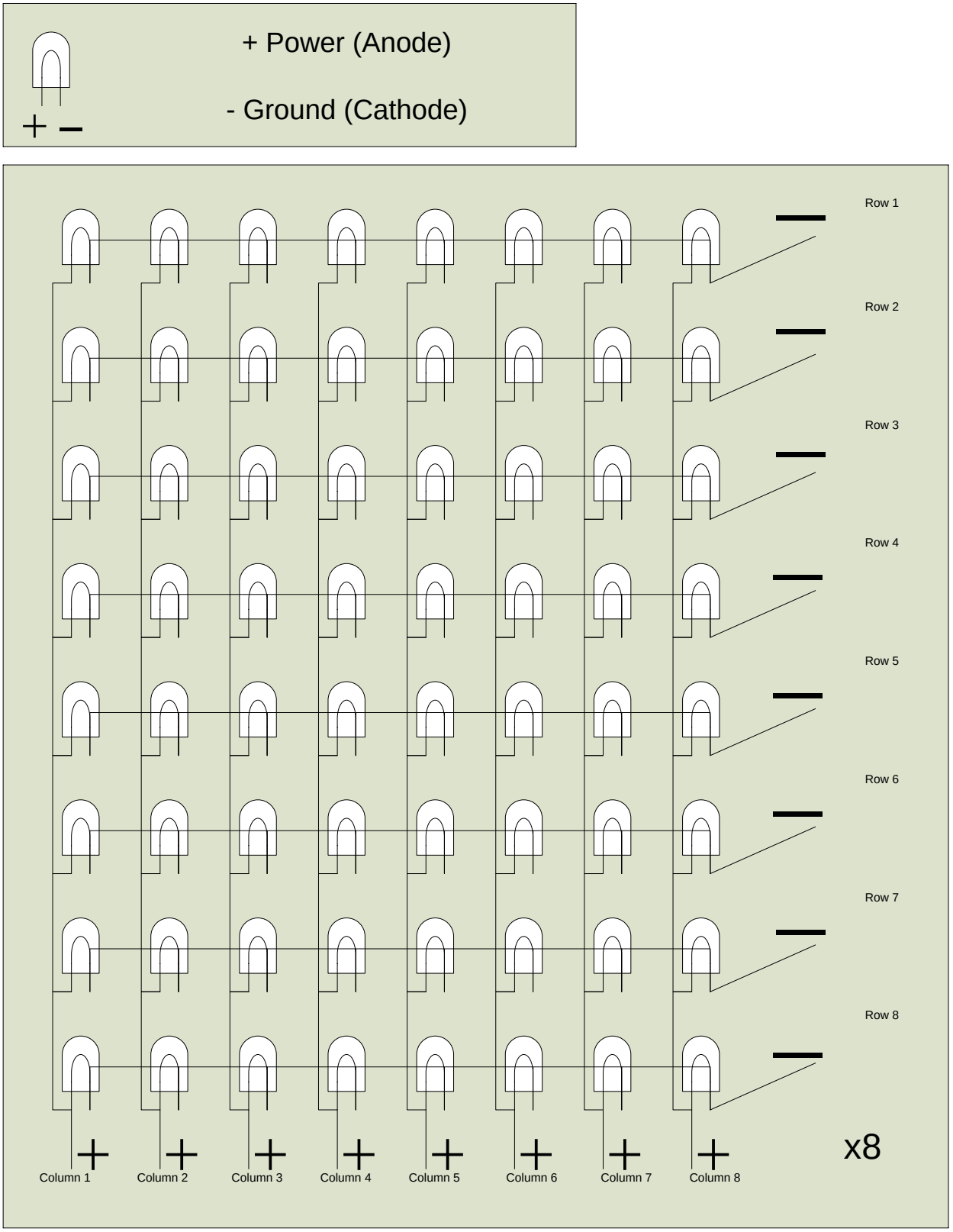
1. Power to Column 1
2. Ground Row 1

To turn on all of Column 1:

1. Power to Column 1
2. Ground Rows 1-8

To turn on all of Row 1:

1. Power to Columns 1-8
2. Ground Row 1


+ Power (Anode)
- Ground (Cathode)
Row 1
Row 2
Row 3
Row 4
Row 5
Row 6
Row 7
Row 8
Column 1
Column 2
Column 3
Column 4
Column 5
Column 6
Column 7
Column 8
x8

LED Driver Hardware Design:

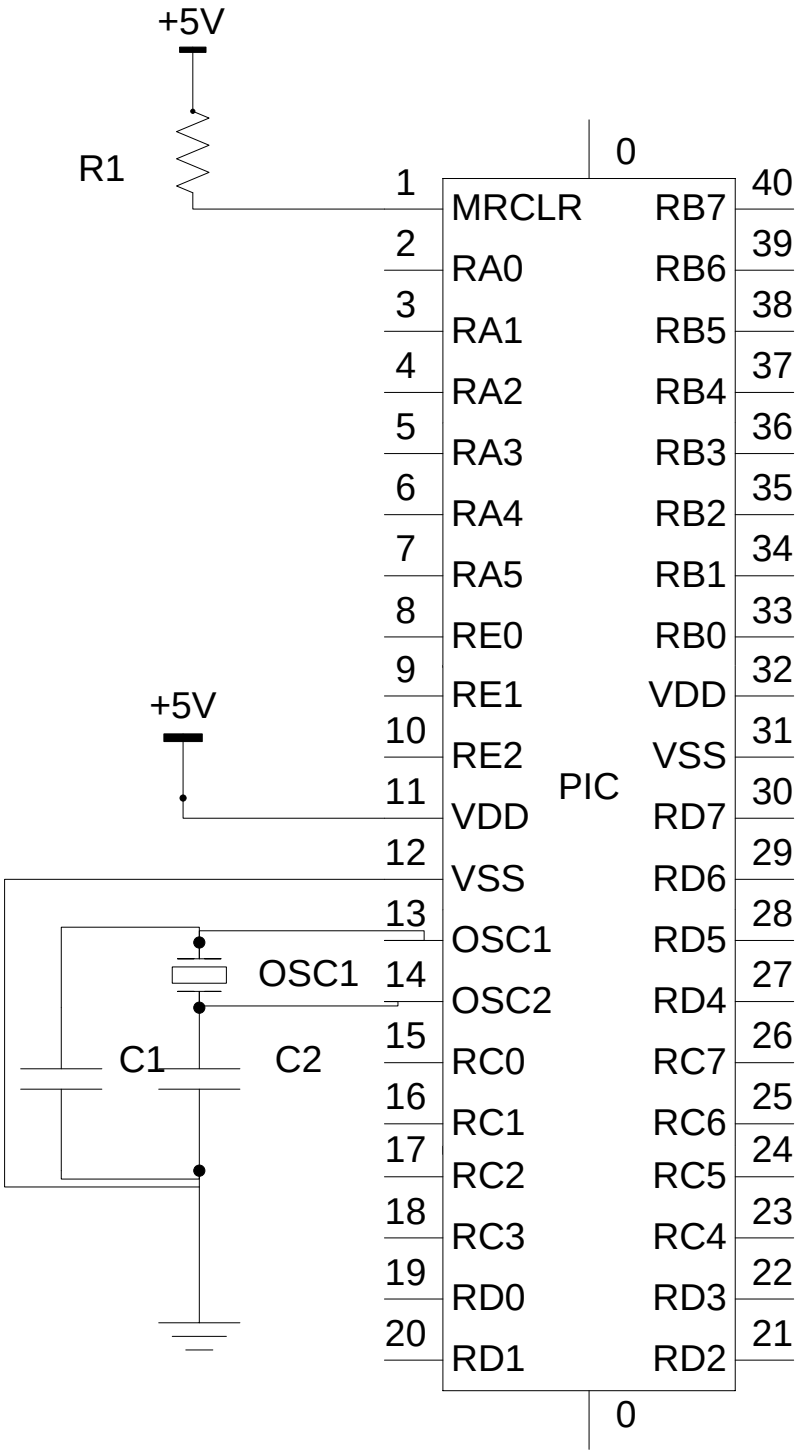
Decision #1: To be able to program, process audio, and control the turn on/off of the LEDs, a processor must be selected. The only functionality needed is to:

1. Be programmable (with use of development board)
2. Read A2D inputs (from audio)
3. Expandable and controllable outputs (for LEDs)
4. Switch modes based on button selection

Selection #1: The **PIC 18F4620** was selected for processor because of ease and knowledge of programming. Other hardware design selections will be based off the use of this processor.

Design #1: (PIC Design)

- **R1:** 10kΩ resistor (based off of datasheet)
    - To limit current from supply
- **C1 and C2:** 22pF capacitors (based off of datasheet)
- **OSC1:** 20MHz oscillator (based off of datasheet)


+5V
R1
+5V
OSC1
C1
C2
PIC
0
1 MRCLR
2 RA0
3 RA1
4 RA2
5 RA3
6 RA4
7 RA5
8 RE0
9 RE1
10 RE2
11 VDD
12 VSS
13 OSC1
14 OSC2
15 RC0
16 RC1
17 RC2
18 RC3
19 RD0
20 RD1
40 RB7
39 RB6
38 RB5
37 RB4
36 RB3
35 RB2
34 RB1
33 RB0
32 VDD
31 VSS
30 RD7
29 RD6
28 RD5
27 RD4
26 RC7
25 RC6
24 RC5
23 RC4
22 RD3
21 RD2
0

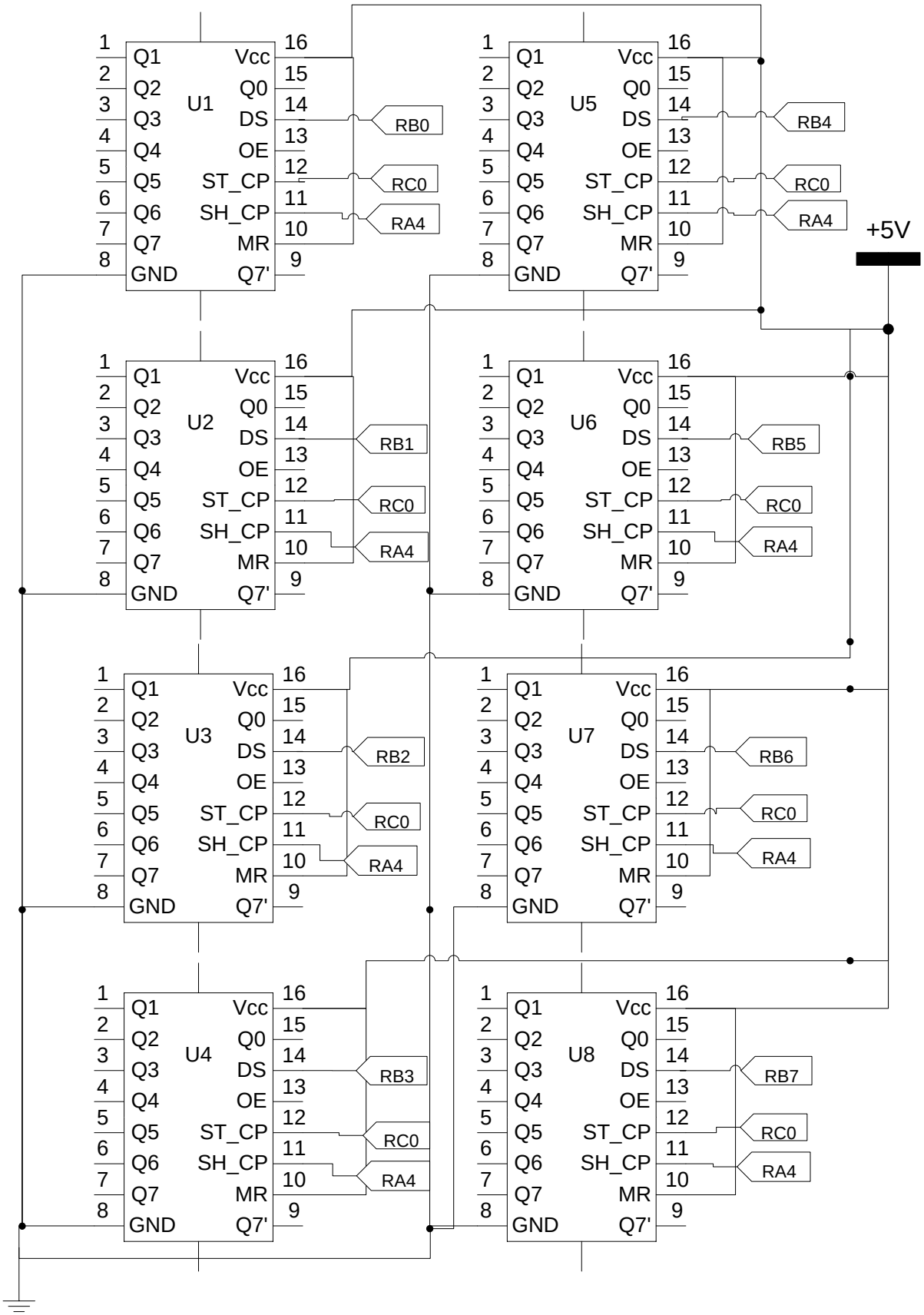
Decision #2: The PIC has a limited number of I/Os. There are not enough outputs to power the LED columns (64 total). Shift registers will be needed to expand the number of outputs to 64. The registers will also need to be controlled to control the LEDs.

Selection #2: The 74HC595N shift registers were chosen to fulfill the needs to expanding the outputs from the PIC to supply necessary 5V power the LED columns.

Design #2:

- **U1-U8:** All are 74HC595N shift registers
  - All Pin 16 connected to 5V supply
  - All Pin 8 connected to ground
  - All Pin 12 (ST_CP) connected to RCO pin on PIC for data clock
  - All Pin 11 (SH_CP) connected to RA4 pin on PIC for shift clock
  - All Pin 10 (Master Reset) pulled high to 5V power
  - All Pin 13/9 are no connection
- **U1:** Pins QO-Q7 are connected to columns 1-8 of LED structure
- **U2:** Pins QO-Q7 are connected to columns 9-16 of LED structure
- **U3:** Pins QO-Q7 are connected to columns 17-24 of LED structure
- **U4:** Pins QO-Q7 are connected to columns 25-32 of LED structure
- **U5:** Pins QO-Q7 are connected to columns 33-40 of LED structure
- **U6:** Pins QO-Q7 are connected to columns 41-48 of LED structure
- **U7:** Pins QO-Q7 are connected to columns 49-56 of LED structure
- **U8:** Pins QO-Q7 are connected to columns 57-64 of LED structure

Note: Explanation of operation examined in software portion.


U1
U2
U3
U4
U5
U6
U7
U8
Q1
Q2
Q3
Q4
Q5
Q6
Q7
GND
Vcc
Q0
DS
OE
ST_CP
SH_CP
MR
Q7'
RB0
RB1
RB2
RB3
RB4
RB5
RB6
RB7
RC0
RA4
+5V

Decision #3: Since the shift registers take care of expanding the outputs and control of the power lines (columns) to the LED structure, extra outputs are available on the PIC for controlling the grounds (rows) of the structure.

Selection #3: In order to control the on/off control of the grounds, a transistor switch from the LED's cathodes (rows) to ground will be used. The transistors will also be needed not only to perform the switching but to sink a large amount of current in case an entire row of LEDs needs to be grounded.

Equations:

$$I_b \times \beta = I_c$$

$$R = \frac{5V - .7}{I_b}$$

Assumed:

$$\beta = 300$$

$$I_{cmax} = 20mA \times 64\ (LEDs\ in\ a\ row) = 1.28A$$

Transistor Selection: TIP31 rated for 3A.

Resistor Selection:

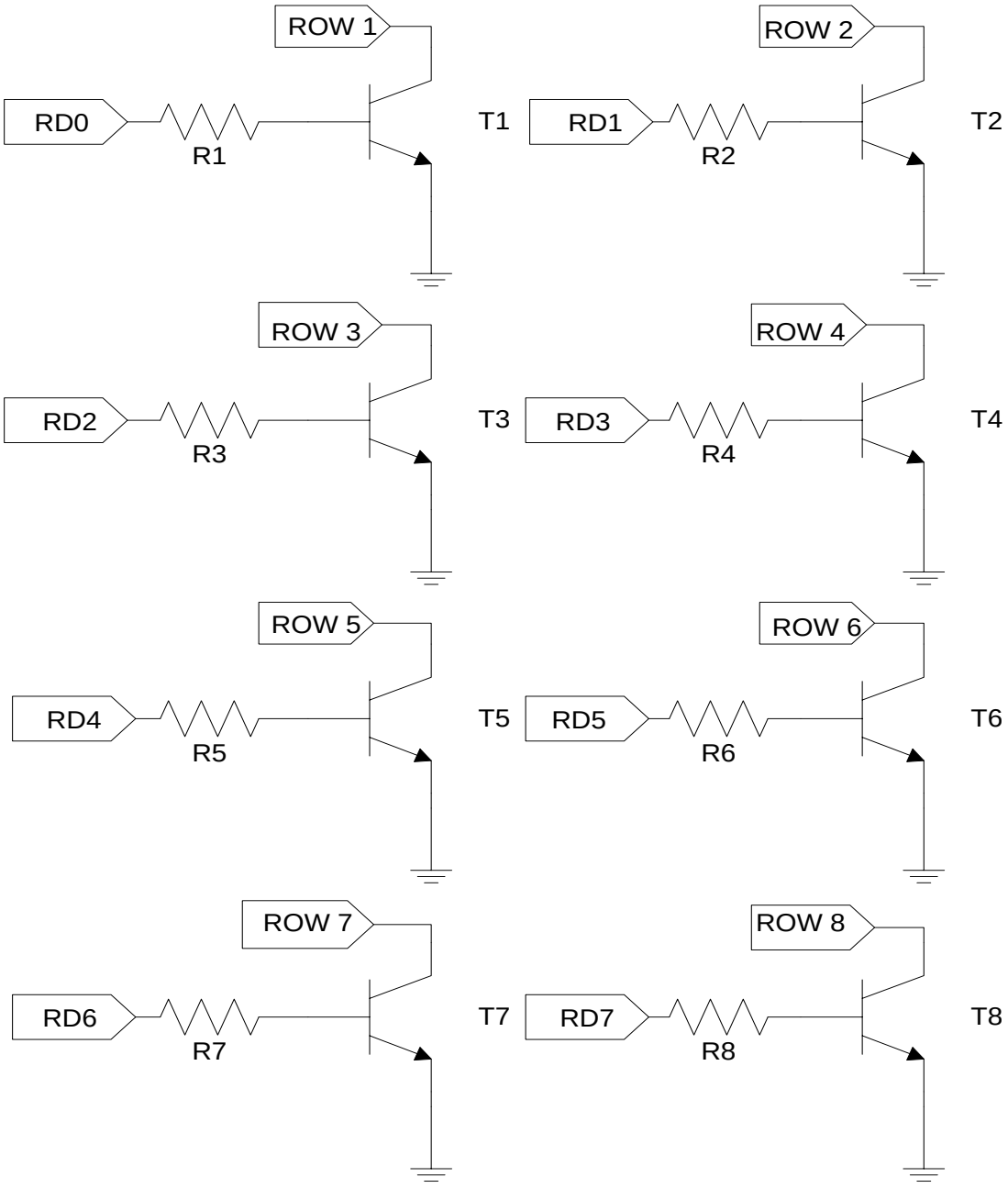
$$I_b = \frac{I_c}{\beta} = \frac{1.28}{300} = 4.3mA$$

$$R = \frac{5V - .7}{I_b} = \frac{4.3}{.0043} = 1000\Omega$$

R=1.1kΩ

Design #4:

- All transistors T1-T8 are TIP31
- All resistors R1-R8 are 1.1kΩ
- RDO-RD7 come from PIC outputs
- Row 1-Row 8 come from the cathodes of the LED structure



Decision#5: In order to switch modes, a button must be used. A signal must be produced being sent to the PIC for the PIC to enable a mode switch.

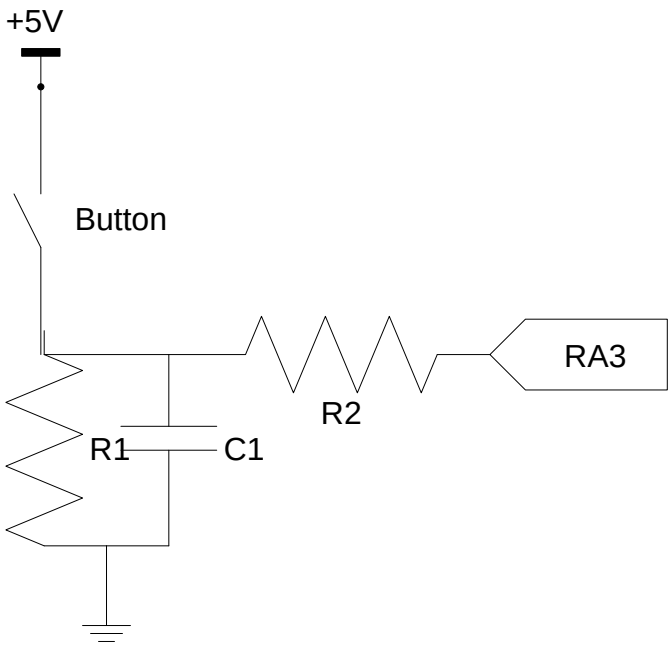
Selection#5: A simple button circuitry will be used.

Design#5:

- A **FSM2JH** button will be used.
- **R1** is a 1.1kΩ resistor to from button to ground to keep the button signal stable and not "bounce". It will also contribute to the RC time constant.
- **C1** a 1µF capacitor is chosen for the RC time constant.
- **R2** is a 100Ω resistor to limit the current going into the PIC.

Equations:

$$\boldsymbol{t(delay) = RC = 1100 \times 10^{-6} = 1.1\ ms}$$

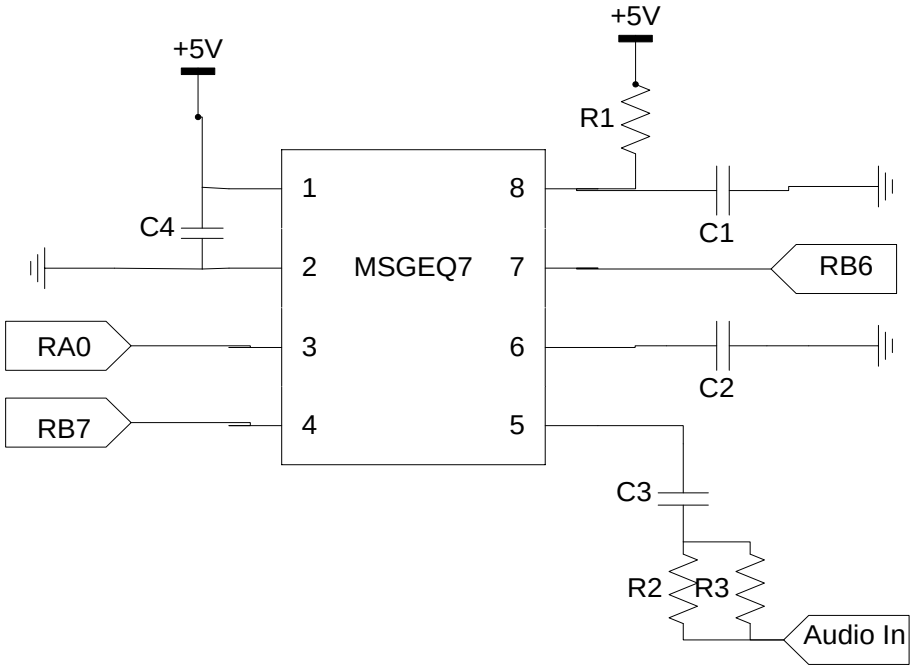


Decision#6: The PIC processor on its own is not capable of reading and processing audio signals. An external chip will be needed to filter out an audio input in order for the PIC to read and then 'project' the filtering into a display on the cube.

Selection#6: A MSGEQ7 Seven Band Graphic Equalizer was chosen. It is capable of splitting up a selected bandwidth of frequencies based on a strobe and reset signal from the PIC. It will then send a single signal to the PIC that will read it as an A2D input. From there the PIC will process the clocks for all of the shift registers and transistors, turning the LEDs on and off.

Design#6:

- All circuit design comes from specifications given in the datasheet.
- The code will be responsible for the operation.
- **R1**=200kΩ
- **R2**=**R3**=22kΩ
- **C1**=33pF
- **C2**=**C4**=0.1µF
- **C3**=0.01 µF



Decision#7: All components in hardware design operate at +5V. To fulfill this power supply need a power supply will be used. This power supply will be able to plug into the wall and supply any power needs to the LED cube circuitry.

Selection#7: A Meanwell Switching Power supply was chosen. It is a plug into wall power supply capable of suppling up to 5A at 5V (25W), plenty of power for our use.

This ends all electrical hardware descriptions. All components in the LED cube were described in the previous decisions. With the seven decisions put together, the resulting circuitry is everything needed to operate the LED cube.

Final BOM:

- 8 x 74HC595N Shift Registers
- 8 x TIP31 Transistors
- 9 x 1.1k resistors
- 1 x PIC184620
- 1 x 20 MHz crystal
- 2 x 22pF capacitors
- 1 x 100k resistor
- 1 x 33pF capacitor
- 2 x 0.01µF capacitor
- 2 x 22k resistor
- 1 x MSGEQ7 IC
- 1 x 0.1µF capacitor
- 1 x FSM2JH switch
- 1 x 25W Meanwell 5V supply
- 512 x Blue 3mm LEDs

Software:

Relying on P.O.V. (persistence of vision) and multiplexing a LED cube can be controlled my connecting the anodes of each layer together and the cathodes of each column together as seen in the Hardware section. To control an individual LED one must power the necessary layer and ground the necessary column for current to flow through the diode. This must be managed across 64 outputs so multiplexers must be used. The multiplexer being used for this project is the 74HC595 with related information from the datasheet shown below.

Table 2. Pin description

Symbol	Pin	Description
Q1	1	parallel data output 1
Q2	2	parallel data output 2
Q3	3	parallel data output 3
Q4	4	parallel data output 4
Q5	5	parallel data output 5
Q6	6	parallel data output 6
Q7	7	parallel data output 7
GND	8	ground (0 V)
Q7S	9	serial data output
$\overline{MR}$	10	master reset (active LOW)
SHCP	11	shift register clock input
STCP	12	storage register clock input
$\overline{OE}$	13	output enable input (active LOW)
DS	14	serial data input
Q0	15	parallel data output 0
V_{CC}	16	supply voltage

7. Functional description

Table 3. Function table[1]

Control				Input	Output		Function
SHCP	STCP	$\overline{OE}$	$\overline{MR}$	DS	Q7S	Qn	
X	X	L	L	X	L	NC	a LOW-level on $\overline{MR}$ only affects the shift registers
X	↑	L	L	X	L	L	empty shift register loaded into storage register
X	X	H	L	X	L	Z	shift register clear; parallel outputs in high-impedance OFF-state
↑	X	L	H	H	Q6S	NC	logic HIGH-level shifted into shift register stage 0. Contents of all shift register stages shifted through, e.g. previous state of stage 6 (internal Q6S) appears on the serial output (Q7S).
X	↑	L	H	X	NC	QnS	contents of shift register stages (internal QnS) are transferred to the storage register and parallel output stages
↑	↑	L	H	X	Q6S	QnS	contents of shift register shifted through; previous contents of the shift register is transferred to the storage register and the parallel output stages

[1] H = HIGH voltage state;
L = LOW voltage state;
↑ = LOW-to-HIGH transition;
X = don't care;
NC = no change;
Z = high-impedance OFF-state.

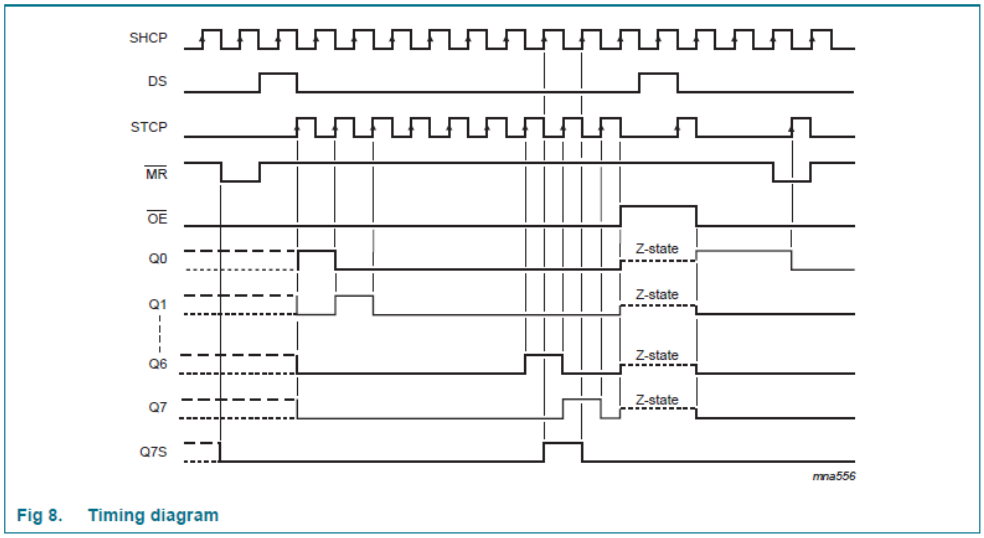
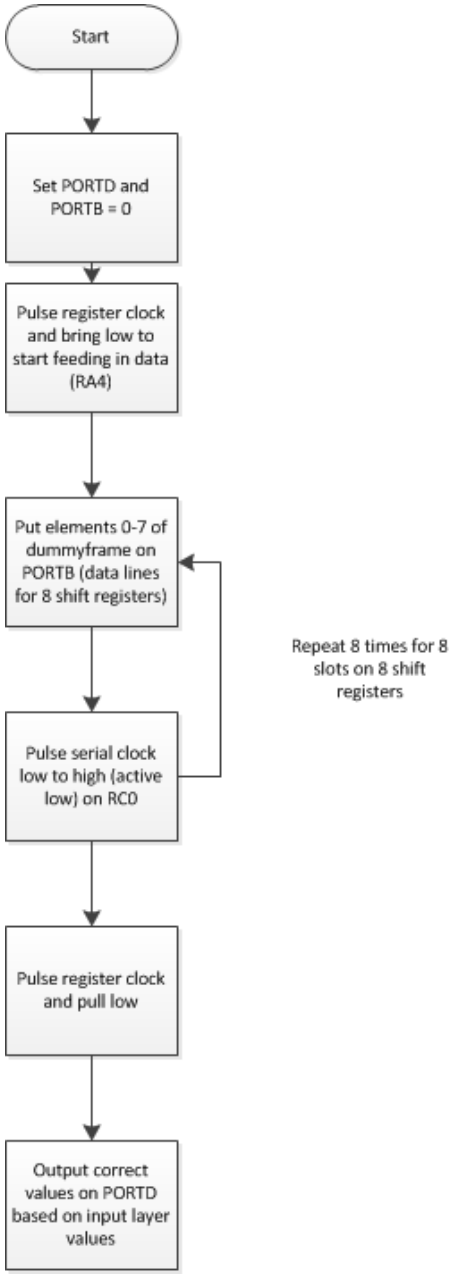


Fig 8. Timing diagram

Assuming we use the following pinouts:

```
//RC0 is master clock for all ports
//PORTD is layers: RD0 = bottom layer, RD1 = next ... RD7 = top
//PORTB is data for columns that need to be clocked out: RB0 = 1st column,
RB1 = 2nd... RB7 = 7th
//RA4 send data clock to regs.
```

This flowchart describes the steps necessary to push the data from the PIC to the cube.



The subroutine that this group came up with looks something like this:

```
void changecolumns(int Layer1, int Layer2, int Layer3, int Layer4, int
Layer5, int Layer6, int Layer7, int Layer8, int wait_time)
{

      //Getting the right row to power for an individual LED
      //Layer = 0 is the off state
      //Layers = 1 -> only RD0 on so PORTD = 00000001;
      //wait_time is the wait between each frame

      //register clock -- RA4 (pin 12) low while data is written. bring high
to output
      //serial data -- All of portB (pin 14)
      //serial clock -- low to high stores bit (pin 11)


      //latch active low ---
      //RC7 = 1;
      PORTD = 0;

      //parsing the data into the column arrays
      // Dane put SendData clock on RA4!!!

      //clear out shift registers before you try anything
      //RA4 = 0;
      PORTB = 0;
      //9 cycles to clear the registers.
      //RC0 = 0;
      /*
RC0 = 1; //1
RC0 = 0;
RC0 = 1; //2
RC0 = 0;
RC0 = 1; //3
RC0 = 0;
RC0 = 1; //4
RC0 = 0;
RC0 = 1; //5
RC0 = 0;
RC0 = 1; //6
RC0 = 0;
RC0 = 1; //7
RC0 = 0;
RC0 = 1; //8
RC0 = 0;
*/
      //output empty register

      //wait a bit to see if something weird happens
      //Wait(500);
      //RC0 = 0;
      //RC0 = 1;
      //RC0 is starting high
      //don't output data yet --> RA4 = 0
      RA4 = 1;
```

```
RA4 = 0;
RB0 = *(dummyframe+0);
RB1 = *(dummyframe+1);
RB2 = *(dummyframe+2);
RB3 = *(dummyframe+3);
RB4 = *(dummyframe+4);
RB5 = *(dummyframe+5);
RB6 = *(dummyframe+6);
RB7 = *(dummyframe+7);
RC0 = 0;
RC0 = 1;
RB0 = *(dummyframe+8);
RB1 = *(dummyframe+9);
RB2 = *(dummyframe+10);
RB3 = *(dummyframe+11);
RB4 = *(dummyframe+12);
RB5 = *(dummyframe+13);
RB6 = *(dummyframe+14);
RB7 = *(dummyframe+15);
RC0 = 0;
RC0 = 1;
RB0 = *(dummyframe+16);
RB1 = *(dummyframe+17);
RB2 = *(dummyframe+18);
RB3 = *(dummyframe+19);
RB4 = *(dummyframe+20);
RB5 = *(dummyframe+21);
RB6 = *(dummyframe+22);
RB7 = *(dummyframe+23);
RC0 = 0;
RC0 = 1;
RB0 = *(dummyframe+24);
RB1 = *(dummyframe+25);
RB2 = *(dummyframe+26);
RB3 = *(dummyframe+27);
RB4 = *(dummyframe+28);
RB5 = *(dummyframe+29);
RB6 = *(dummyframe+30);
RB7 = *(dummyframe+31);
RC0 = 0;
RC0 = 1;
RB0 = *(dummyframe+32);
RB1 = *(dummyframe+33);
RB2 = *(dummyframe+34);
RB3 = *(dummyframe+35);
RB4 = *(dummyframe+36);
RB5 = *(dummyframe+37);
RB6 = *(dummyframe+38);
RB7 = *(dummyframe+39);
RC0 = 0;
RC0 = 1;
RB0 = *(dummyframe+40);
RB1 = *(dummyframe+41);
RB2 = *(dummyframe+42);
RB3 = *(dummyframe+43);
RB4 = *(dummyframe+44);
```

```
	RB5 = *(dummyframe+45);
	RB6 = *(dummyframe+46);
	RB7 = *(dummyframe+47);
	RC0 = 0;
	RC0 = 1;
	RB0 = *(dummyframe+48);
	RB1 = *(dummyframe+49);
	RB2 = *(dummyframe+50);
	RB3 = *(dummyframe+51);
	RB4 = *(dummyframe+52);
	RB5 = *(dummyframe+53);
	RB6 = *(dummyframe+54);
	RB7 = *(dummyframe+55);
	RC0 = 0;
	RC0 = 1;
	RB0 = *(dummyframe+56);
	RB1 = *(dummyframe+57);
	RB2 = *(dummyframe+58);
	RB3 = *(dummyframe+59);
	RB4 = *(dummyframe+60);
	RB5 = *(dummyframe+61);
	RB6 = *(dummyframe+62);
	RB7 = *(dummyframe+63);
	RC0 = 0;
	RC0 = 1;
	//now output
	RA4 = 1;
	RA4 = 0;


	//parse layer information and turn on the right rows. RD0-->Bottom
layer , RD7 --> Top layer


	//calculate correct value for PORTD
	//PORTD = (Layer1*1) + (Layer2*2) + (Layer3*4) + (Layer4*8) +
(Layer5*16) + (Layer6*32) + (Layer7*64) + (Layer8*128);

	if (Layer1 == 1){RD0 = 1;} else RD0 = 0;
	if (Layer2 == 1){RD1 = 1;} else RD1 = 0;
	if (Layer3 == 1){RD2 = 1;} else RD2 = 0;
	if (Layer4 == 1){RD3 = 1;} else RD3 = 0;
	if (Layer5 == 1){RD4 = 1;} else RD4 = 0;
	if (Layer6 == 1){RD5 = 1;} else RD5 = 0;
	if (Layer7 == 1){RD6 = 1;} else RD6 = 0;
	if (Layer8 == 1){RD7 = 1;} else RD7 = 0;

	Wait(wait_time);
	//Wait(10); //test

}
```

Note how a global variable "dummy frame" is parsed out to the data lines in the middle of this piece of code. Having this global variable makes it easy for any program to edit it and call the changecolumns routine. Editing the dummyframe and calling the changecolumns routine gives you complete control of the cube and animate seemingly any object represent able on a 8x8x8 cube. It also helps to write the dummyframe like this:

```
int dummyframe[64] = {
      0,0,0,0,0,0,0,0,
      0,0,0,0,0,0,0,0,
      0,0,0,0,0,0,0,0,
      0,0,0,0,0,0,0,0,
      0,0,0,0,0,0,0,0,
      0,0,0,0,0,0,0,0,
      0,0,0,0,0,0,0,0,
      0,0,0,0,0,0,0,0,
};
```

Think of it as a layer from looking above the cube where you have to light the correct LED's in the frame (turn the 0's to 1's) and turn on the necessary levels with the changecolumns routine. Now that you can control the LED's cube you can have other routines, sensors, inputs trigger editing the dummyframe and calling the changecolumns routine. The possibilities are endless.

Shape/Animation Example: "Drawing a cube"

```
void cube2(int time)
{
      for (int p_c2 = 0; p_c2<64; p_c2++)
      {
            dummyframe[p_c2] = 0;
      }
      dummyframe[18] = 1;
      dummyframe[19] = 1;
      dummyframe[20] = 1;
      dummyframe[21] = 1;
      dummyframe[26] = 1;
      dummyframe[29] = 1;
      dummyframe[34] = 1;
      dummyframe[37] = 1;
      dummyframe[42] = 1;
      dummyframe[43] = 1;
      dummyframe[44] = 1;
      dummyframe[45] = 1;
      changecolumns(0,0,1,1,1,1,0,0,time);
}
```

Notice how the dummyframe is initially cleared with a "for" loop then filled with values to make a box.

```
dummyframe[64] = {
      0,0,0,0,0,0,0,0,
      0,0,0,0,0,0,0,0,
      0,0,1,1,1,1,0,0,
      0,0,1,0,0,1,0,0,
      0,0,1,0,0,1,0,0,
      0,0,1,1,1,1,0,0,
      0,0,0,0,0,0,0,0,
      0,0,0,0,0,0,0,0,
};
```

Then changecolumns is called with the middle 4 layers lit up. In this way this new subroutine "cube2" is the only thing that needs to be called to draw this size of box on the LED cube. Other shapes can be done in a similar fashion and made into separate subroutines for ease of use.

Audio Chip code: In a similar manner to the way the code was written for the shift registers in the "changecolumns" routine an "audioread" subroutine was constructed to control the audio chip(MSGEQ7). This was done by simply referring to the datasheet.

```
void audio_read(void)
{
      unsigned int amp[8];
      unsigned int amp_i;
      for (amp_i=0; amp_i<8; amp_i++)
      {
            if (amp_i == 2){
                  int fuzzy_logic =
(int)(spectrumRead[1]*.5+spectrumRead[3]*.5+.5);
                  spectrumRead[2] = fuzzy_logic; //artificial value
            }else{
                  RA2 = 0; //Strobe LOW
                  Wait(1); // Lets the filter settle
                  amp[amp_i]= A2D_Read(0);
                  if(amp[amp_i]>1010){spectrumRead[amp_i] = 8; silent_count =
0;}
                  else if(amp[amp_i]>975) {spectrumRead[amp_i] = 7;
silent_count = 0;}
                  else if(amp[amp_i]>930) {spectrumRead[amp_i] = 6;
silent_count = 0;}
                  else if(amp[amp_i]>730) {spectrumRead[amp_i] = 5;
silent_count = 0;}
                  else if(amp[amp_i]>530) {spectrumRead[amp_i] = 4;
silent_count = 0;}
                  else if(amp[amp_i]>430) {spectrumRead[amp_i] = 3;
silent_count = 0;}
                  else if(amp[amp_i]>290) {spectrumRead[amp_i] = 2;
silent_count = 0;}
                  else if(amp[amp_i]>100) {spectrumRead[amp_i] = 1;}
                  else {spectrumRead[amp_i] = 0; silent_count =
silent_count++;} //increment if no audio found

                  RA2 = 1; //Strobe HIGH
                  Wait(1); //Might not be needed
            }
      }
}
```

Note how the main portion of code is filling an arrary "amp[8]" with values read in through the A2D port. Since the audio chip has a much higher clock than the PIC we don't have to worry about reading the data too soon and can easily fill as many arrays as necessary simply by strobing RA2 (which is tied to the strobe pin of the audio chip). These values are how the cube can "react" to the music.

Mode Code: The LED cube had different “modes” that can be switched through RC7. This is done with a simple interrupt Timer2 interrupt :

```
void interrupt IntServe(void)
{
      // Is RC1 button being pushed? Then change modes.
      if (CCP2IF) {
            mode = mode+1;
            //Up to 5 modes
            if (mode > 4) mode = 0;
            CCP2IF = 0;
      }
}
```

It is quite easy to see that we can now have a “while(1)” loop in our “main” loop that looks something like this:

```
while(1) {
            while(mode == 0)
            {
                  ***react to amp[1], amp[2], amp[3] from audioread*****

                  if(silent_count>5000) mode = 1; //auto next mode if noaudio
            }
            while(mode == 1)
            {
                  **react to entire amp[8] array from audio read**
            }
            while(mode == 2)
            {
                  **Animate a sparkle pattern**
            }
```

and so on… In this pseudocode one can easily see that using the subroutines described above one can construct a LED cube that can react to music.

Technician's Troubleshooting Section:

Complete the following actions when the Led Cube does not function correctly.

Action #1: Make sure you have the device plugged in all the way and it is connected to a working outlet.

Action #2: Make sure that the connection between the external music source and the jack is connected.

Action #3: Try pressing the "Mode Change" button on the side of the LED Cube multiple times.

Action #4: Unplug the LED Cube from the wall outlet and replug.

WARNINGS:

Take caution when using the LED Cube.

Disconnect the AC wall transformer before cleaning the LED Cube.

Do not leave the LED Cube plugged in when not in use.

Staring at the Cube may cause instant blindness.

Do not submerge the unit in water, as it will damage the electrical components.

Do not operate LED Cube without safety case covering device.

Do not touch the LEDs when powered on.

Project Comments:

Testing and Evaluation

- Put heartbeat to see if portions of the code is working.
- Create Matlab scripts to create an audio spectrum sweep
- Turn on all LED's on the cube to see if connections are good
- See if visually it "bounced" to the music
- Isn't affected by volume of the output speaker
- Tested connections continuously with a multimeter
- Bottom-up programming –starting small and building up piece by piece
- Breakout files to easily add or subtract pieces of code
- Realizing the LED's were dimmer than expected
- Trying new things and knowing when to quit trying to get it to work and move to something that can work.
- Prototyping the cube
- Equipment

Problems Encountered

- We found our shift registers would store values from previously ran code which caused glitches and errors in our mode switching.
- Out of scope errors
- 1) Debugging with optimizations on can make variables appear out of scope (obviously)
- 2) Volatile does not necessarily fix the out of scope problem, at least for the attempted local variables
- 3) "Isolate each function in a section" option which passes -functions-sections option to the compiler may alleviate the out of scope problem.
- Feedback from audio circuit
- Cheap speakers
- How to clock a shift register
- Bad chips
- incorrectly reading datasheets
- Dangerous equipment
- Is it getting power?
- Ordering PCB's
- Rods/Structure
- Always trying to change the project
- Backing up code

- Connections

Lesson Learned

- Aim low so that you can hit your mark
- Document stuff better
- Get everyone on the same page
- Plan modularly
- Don't try to juggle a full time co-op with senior design
- Seemingly simple things can make for complex problems

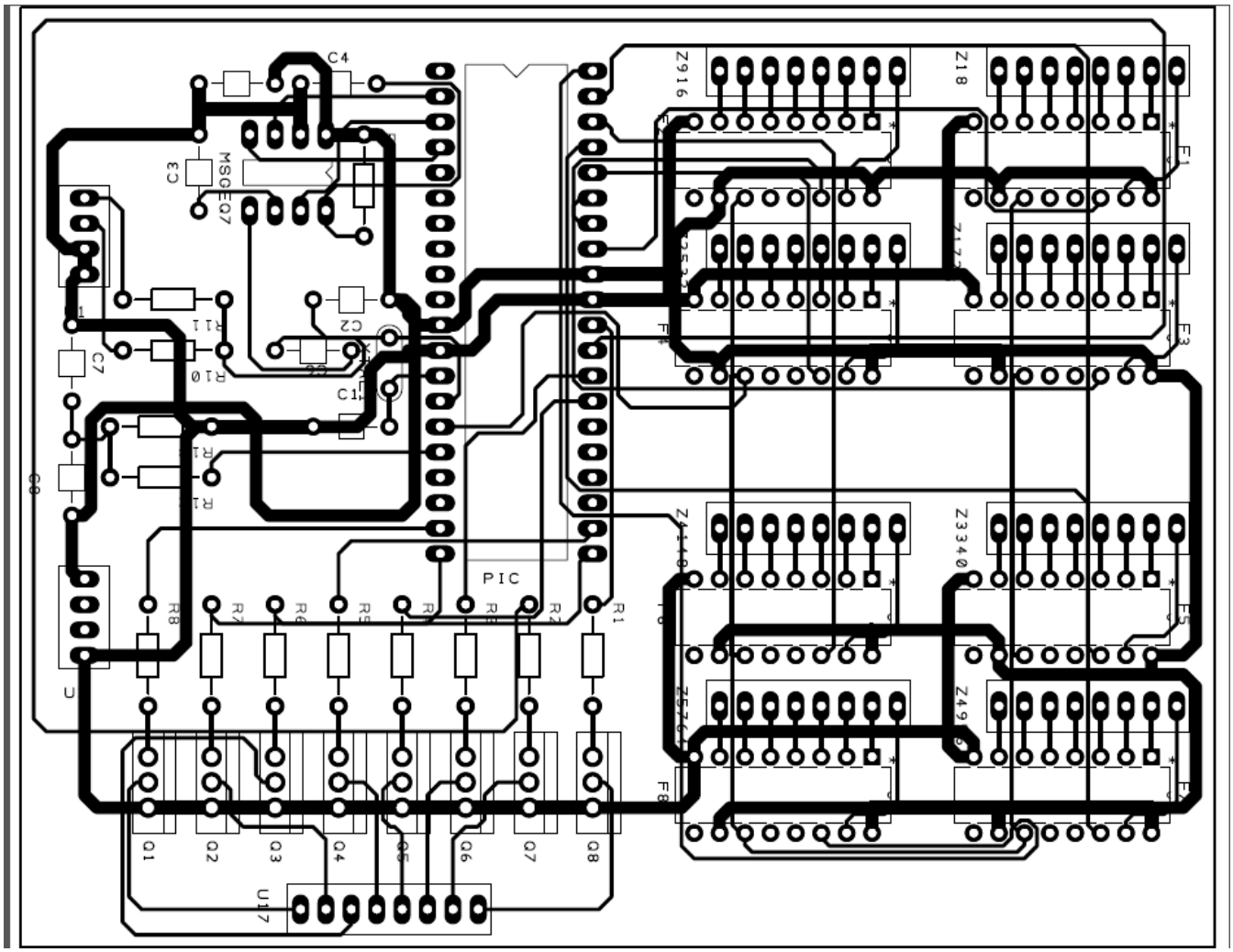
BUDGET:

Component	Estimated	Price	Total
100 LEDS	$21	Free	Free
500 LEDS	$250	$40	$40
Mounting Supplies	$20	Free	Free
Audio Processor	$150	$20	$20
3.5mm Audio Jack	$5	Free	Free
Block of Wood	Free	Free	Free
Electrical Parts	$50	$20	$20
Housing	$25	$40	$40
MP3 Player	$15	$15	Free
Keyboard (Piano)	$200	Free	Free
Speaker	$40	Free	Free
Outsource PCB	$60	$50	$50
Super Glue	$4	$4	$4
Rubber Cement	$5	Free	Free
Spray Paint	$4	$4	$4
PIC Processor Components	$99	Free	Free
Audio Splitter	$10	Free	Free

Budget
Final Total: $178

Appendix:

PCB Layout:



7 Band Equalizer Datasheet:

9/2004

msi

Seven Band Graphic Equalizer Data Sheet

MSGEQ7

Description

The seven band graphic equalizer IC is a CMOS chip that divides the audio spectrum into seven bands, 63Hz, 160Hz, 400Hz, 1kHz, 2.5kHz, 6.25kHz and 16kHz. The seven frequencies are peak detected and multiplexed to the output to provide a DC representation of the amplitude of each band. No external components are needed to select the filter responses. Only an off chip resistor and capacitor are needed to select the on chip clock oscillator frequency. The filter center frequencies track this frequency.

Other than coupling and decoupling capacitors, no other external components are needed. The chip supply can be between 2.7 and 5.5 volts with 5 volts providing the best performance. The device has very low quiescent current (less than 1mA typical) for portable audio devices. The multiplexor is controlled by a reset and a strobe, permitting multiplexor readout with only two pins. The multiplexor readout rate also controls the decay time (10% decay per read), so no external pins are needed for this function.

Features

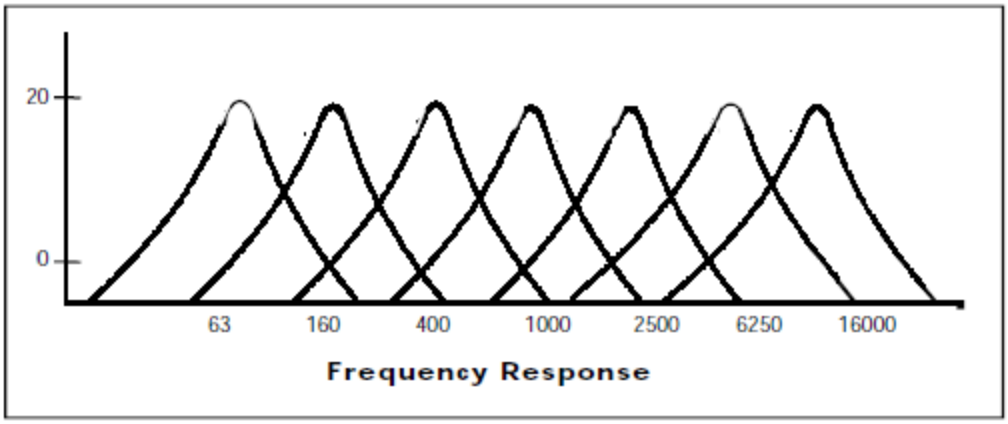
Low Power Consumption
Only Two External Components
On Chip Ground Reference
Switched - Capacitor Filters
3.3 or 5 volt Operation
20 dB of Gain Typical
On Chip Oscillator
Output Multiplexor
Variable Decay Time
8 Pin Package

Applications

Portable Stereos
Car Stereos
Hi-Fi Stereos
Spectrum Analyzers

Absolute Maximum Ratings

Power Supply Voltage	+6V
Storage Temperature	-60 to + 150 C
Operating Temperature	0 to 70 C



msi

Web Site "www.mix-sig.com"

 1

Shift Register Datasheet:

74HC595; 74HCT595

8-bit serial-in, serial or parallel-out shift register with output latches; 3-state

Rev. 6 — 12 December 2011 Product data sheet

1. General description

The 74HC595; 74HCT595 are high-speed Si-gate CMOS devices and are pin compatible with Low-power Schottky TTL (LSTTL). They are specified in compliance with JEDEC standard No. 7A.

The 74HC595; 74HCT595 are 8-stage serial shift registers with a storage register and 3-state outputs. The registers have separate clocks.

Data is shifted on the positive-going transitions of the shift register clock input (SHCP). The data in each register is transferred to the storage register on a positive-going transition of the storage register clock input (STCP). If both clocks are connected together, the shift register will always be one clock pulse ahead of the storage register.

The shift register has a serial input (DS) and a serial standard output (Q7S) for cascading. It is also provided with asynchronous reset (active LOW) for all 8 shift register stages. The storage register has 8 parallel 3-state bus driver outputs. Data in the storage register appears at the output whenever the output enable input ($\overline{OE}$) is LOW.

2. Features and benefits

- 8-bit serial input
- 8-bit serial or parallel output
- Storage register with 3-state outputs
- Shift register with direct clear
- 100 MHz (typical) shift out frequency
- ESD protection:
  - HBM JESD22-A114F exceeds 2000 V
  - MM JESD22-A115-A exceeds 200 V
- Multiple package options
- Specified from −40 °C to +85 °C and from −40 °C to +125 °C

3. Applications

- Serial-to-parallel data conversion
- Remote control holding register

PIC Microcontroller Datasheet:

MICROCHIP **PIC18F2525/2620/4525/4620**

28/40/44-Pin Enhanced Flash Microcontrollers with 10-Bit A/D and nanoWatt Technology

Power Management Features:

- Run: CPU on, Peripherals on
- Idle: CPU off, Peripherals on
- Sleep: CPU off, Peripherals off
- Ultra Low 50nA Input Leakage
- Run mode Currents Down to 11 µA Typical
- Idle mode Currents Down to 2.5 µA Typical
- Sleep mode Current Down to 100 nA Typical
- Timer1 Oscillator: 900 nA, 32 kHz, 2V
- Watchdog Timer: 1.4 µA, 2V Typical
- Two-Speed Oscillator Start-up

Flexible Oscillator Structure:

- Four Crystal modes, up to 40 MHz
- 4x Phase Lock Loop (PLL) – Available for Crystal and Internal Oscillators
- Two External RC modes, up to 4 MHz
- Two External Clock modes, up to 40 MHz
- Internal Oscillator Block:
  - Fast wake from Sleep and Idle, 1 µs typical
  - 8 use-selectable frequencies, from 31 kHz to 8 MHz
  - Provides a complete range of clock speeds from 31 kHz to 32 MHz when used with PLL
  - User-tunable to compensate for frequency drift
- Secondary Oscillator using Timer1 @ 32 kHz
- Fail-Safe Clock Monitor:
  - Allows for safe shutdown if peripheral clock stops

Peripheral Highlights:

- High-Current Sink/Source 25 mA/25 mA
- Three Programmable External Interrupts
- Four Input Change Interrupts
- Up to 2 Capture/Compare/PWM (CCP) modules, one with Auto-Shutdown (28-pin devices)
- Enhanced Capture/Compare/PWM (ECCP) module (40/44-pin devices only):
  - One, two or four PWM outputs
  - Selectable polarity
  - Programmable dead time
  - Auto-shutdown and auto-restart

Peripheral Highlights (Continued):

- Master Synchronous Serial Port (MSSP) module Supporting 3-Wire SPI (all 4 modes) and I^2C™ Master and Slave modes
- Enhanced Addressable USART module:
  - Supports RS-485, RS-232 and LIN/J2602
  - RS-232 operation using internal oscillator block (no external crystal required)
  - Auto-wake-up on Start bit
  - Auto-Baud Detect
- 10-Bit, up to 13-Channel Analog-to-Digital (A/D) Converter module:
  - Auto-acquisition capability
  - Conversion available during Sleep
- Dual Analog Comparators with Input Multiplexing
- Programmable 16-Level High/Low-Voltage Detection (HLVD) module:
  - Supports interrupt on High/Low-Voltage Detection

Special Microcontroller Features:

- C Compiler Optimized Architecture:
  - Optional extended instruction set designed to optimize re-entrant code
- 100,000 Erase/Write Cycle Enhanced Flash Program Memory Typical
- 1,000,000 Erase/Write Cycle Data EEPROM Memory Typical
- Flash/Data EEPROM Retention: 100 Years Typical
- Self-Programmable under Software Control
- Priority Levels for Interrupts
- 8 x 8 Single-Cycle Hardware Multiplier
- Extended Watchdog Timer (WDT):
  - Programmable period from 4 ms to 131s
- Single-Supply 5V In-Circuit Serial Programming™ (ICSP™) via Two Pins
- In-Circuit Debug (ICD) via Two Pins
- Wide Operating Voltage Range: 2.0V to 5.5V
- Programmable Brown-out Reset (BOR) with Software Enable Option

Device	Program Memory		Data Memory		I/O	10-Bit A/D (ch)	CCP/ ECCP (PWM)	MSSP		EUSART	Comp.	Timers 8/16-Bit
	Flash (bytes)	# Single-Word Instructions	SRAM (bytes)	EEPROM (bytes)				SPI	Master I^2C™			
PIC18F2525	48K	24576	3968	1024	25	10	2/0	Y	Y	1	2	1/3
PIC18F2620	64K	32768	3968	1024	25	10	2/0	Y	Y	1	2	1/3
PIC18F4525	48K	24576	3968	1024	36	13	1/1	Y	Y	1	2	1/3
PIC18F4620	64K	32768	3968	1024	36	13	1/1	Y	Y	1	2	1/3

